Documentation of Project Implementation for IPP 2017/2018
Name and surname: Anton Firc
Login: xfirca00

parse.php

Script for lexical and syntactical analysis of source code in IPPcode18. Gets input from standard input and outputs formatted representation of source code to standard output in XML format.

Script can be run with `--help`(brief purpose and usage information) parameter. Parameters are parsed using builtin function `getopt()`.

Code analysis starts with checking for header using `parse_header` function. Every read line is stripped of blank characters and all blank lines and lines including only commentary are ignored. Function returns count of comments found before and on header line. After header is found, XML file is created and correctly formatted. Analysis runs in infinite loop which is explicitly broken by reading EOF character. Every line of source code is analysed separately and creates single instruction attribute in XML file. Instructions are divided into groups by count and type of arguments. Each group has its set of rules that are applied, and if instruction does not meet those rules analysis exits with a failure. After processing all instructions (EOF is read), the XML file is closed correctly and script exits.

interpret.py

Script for interpreting programs in IPPcode18. Gets input from XML file generated by parse.php script and executes all instructions.

Script can be run with `--help`(brief purpose and usage information) and `--source="file"`(source XML file / parse.php output) parameters. Parameters are processed using `argparse`.

Interpret starts with loading all instructions into list, using class `Instruction` that holds information about instruction opcode, order and list of arguments. Argument of instruction has its own class that holds information about type (var, int, string, bool) and text (variable name, constant). Special attention is given to labels. When label is found, it is immediately processed. Label name and order are stored in object that is later stored in list of labels. Processing labels when loading code is necessary because of `JUMP` instructions which can jump „forward“ in code. After all instructions are loaded, while loop cycles through them in order and executes them. Variables are stored in two-dimensional array according to memory frame that is currently active. Default memory frame is `Global` and further frame switching is secured by `CREATEFRAME, PUSHFRAME` and `POPFRAME` instructions. Each variable is an object that holds name, value and memory frame. Jump instructions use while cycle index, they change index to array of loaded instructions to move to label location. The same applies to `CALL` instruction, which in addition stores order of next instruction to `call_stack` and jumps to label. After calling `RETURN` the last value in `call_stack` is popped and used as index to instructions array. Interpretation ends when last instruction is executed (index to instructions array is same as count of instructions stored).

test.php

Testing unit for parse.php and interpret.py scripts.

Script can be run with `--help`(brief purpose and usage information), `--directory="path"`(optional, path to tests diretory), `--recursive`(optional, recursively search subdirectories for tests), `--parse-script="file"`(parse.php file) and `--int-script="file"`(interpret.py file). Parameters are processed using builtin function `getopt()`.

On startup script processes all parameters, if `--directory` parameter is not used, script uses current work directory. Output HTML file is opened and header is generated. Function `opensubdirs` then scans directory for test folders. If folder contains files needed for testing is decided by `istest` function which scans given directory for `.src, .out, .in` and `.rc` file. If all files are present folder is marked as test directory. In other case if parameter `--recursive` is specified, `opensubdirs` functon is called and scans subdirectories. Function `run_test` opens test directory and sequentially runs parse.php and interpret.py scripts and compares return code. Output is saved to temporary file `result`. If return code matches `.rc` file, `result` file is compared to `.out` file using `diff` (bash builtin tool). If `diff` exits with code 0, files are identical and test ends with pass. After test completion result is written to HTML file(path to test : PASS/FAIL). Script keeps track of executed and passed test and after completing all test prints brief summary. Summary contains count of succesful tests, all tests and pass percentage.